Randomized Testing Framework for Dissecting NVIDIA GPGPU Thread Block-To-SM Scheduling Mechanisms

Chongxi Wang[1,2], Penghao Song[1,2], Haoyu Zhao[1,2], Fuxin Zhang[1,2], Jian Wang[1,2], and Longbing Zhang[1,2]

[1]*State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China*
[2]*University of Chinese Academy of Sciences, Beijing, China*
{wangzhongxi15, songpenghao16}@mails.ucas.ac.cn, {zhaohaoyu19s, fxzhang, jw, lbzhang}@ict.ac.cn

***Abstract*—NVIDIA General Purpose Graphics Processing Units (GPGPUs) have been extensively utilized in scientific computing, artificial intelligence acceleration, and numerous other fields. The compute unified device architecture (CUDA) programming interface facilitates access to the enormous computing power of NVIDIA GPGPUs. However, the thread block-to-stream multiprocessor (SM) scheduler of NVIDIA GPGPUs has remained a black box, presenting challenges for performance modeling of CUDA programs, especially those featuring concurrent kernel execution, as well as hindering further exploration of the full potential of NVIDIA GPGPUs. To address these issues, we have devised a randomized testing framework targeting thread block-to-SM scheduling. In this framework, concurrent kernel sequences are randomly generated and assessed by a self-implemented scheduling predictor along with an actual NVIDIA GPGPU. Through iteratively scrutinizing the discrepancies between predictions and actual results, we dissect the key thread block-to-SM scheduling mechanisms intrinsic to NVIDIA GPGPUs, revealing their underlying logic and considerations. This process continuously refines the scheduling predictor until it accurately aligns with real-world NVIDIA GPGPU scheduling. Furthermore, we investigate the monitoring and allocation mechanisms for distinct resources in NVIDIA GPGPUs and SMs, elucidating the essential connection between resource management and thread block scheduling. Our comprehensive and precise analysis of Nvidia GPGPU thread block-to-SM scheduling contributes significantly to constructing more accurate simulators for GPGPU performance modeling. Moreover, it facilitates optimizing CUDA programs based on the scheduling mechanisms to exploit more parallelism among thread blocks of concurrent kernels.**

***Index Terms*—Thread block-to-SM scheduling, randomized testing framework, resource management, NVIDIA GPGPUs**

I. Introduction

In recent years, NVIDIA General Purpose Graphics Processing Units (GPGPUs) have become one of the most sought-after chip products. Since the launch of the compute unified device architecture (CUDA) programming interface in 2006 [1], NVIDIA GPGPUs have gradually demonstrated their capabilities in general computing. Leveraging the gaming market and the ongoing artificial intelligence (AI) boom that began in the 2010s [2]–[5], NVIDIA GPGPUs have become an indispensable computing platform from household use to data centers. With the continuous evolution of their architecture, NVIDIA GPGPUs provide higher computational power and richer programming features. More and more applications are accelerating with CUDA programming, achieving significant acceleration effects.

NVIDIA GPGPUs adopt the SIMT architecture and programming model, allowing for concurrent execution of threads, which is the root of the scalability of GPGPU computing power. In the CUDA programming model, threads are organized into thread blocks and scheduled to execute on stream multiprocessors (SMs). The CUDA stream mechanism enables the concurrent execution of different kernels. Based on GPGPU multitasking methods [6], works focusing on multitasking parallelism have achieved notable results [7]–[11]. As the computational, storage, and memory access resources within each SM are limited, there exists resource complementarity or resource conflict among different thread blocks. Therefore, the optimization of multitasking parallelism is closely related to thread block scheduling. Works focusing on scheduling strategies have also achieved promising performance improvements [12]–[17].

The thread block-to-SM scheduling mechanisms of NVIDIA GPGPUs have long remained a black box. Works focusing on scheduling optimization typically use round-robin as a baseline [12], [15]. According to the previous analyses [18], [19], NVIDIA GPGPUs utilize scheduling algorithms that are more conducive to SM load balancing compared to the simple round-robin approach. These scheduling mechanisms involve monitoring inactive resources in all SMs to make reasonable thread block scheduling decisions. Nevertheless, these studies have only provided insights through a limited number of rudimentary examples, lacking comprehensive validation. The thread block-to-SM scheduling in NVIDIA GPGPUs is closely intertwined with the resource management mechanisms of the SMs. These studies fall short of thoroughly examining the various SM resource management mechanisms. Consequently, the scheduling algorithms they propose may deviate from the actual scheduling results of NVIDIA GPGPUs in many scenarios and are incapable of

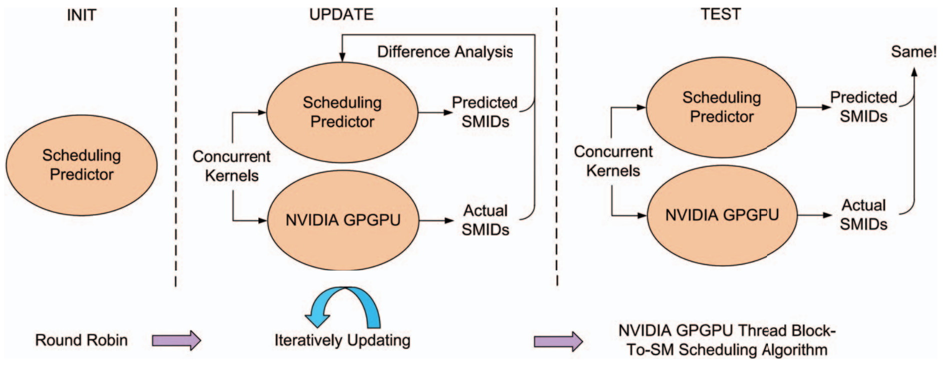

Fig. 1. Overall research process of dissecting NVIDIA GPGPU thread block-to-SM scheduling mechanisms.

passing the randomized tests devised in this work.

Building upon the previous studies, this work presents a comprehensive analysis of NVIDIA GPGPU thread block-to-SM scheduling mechanisms and the underlying SM resource monitoring and allocation mechanisms. The derived thread block-to-SM scheduling algorithm, when subjected to complex randomly generated concurrent kernel sequences, remains consistent with the actual scheduling results of NVIDIA GPGPUs. The overall research process is depicted in Fig. 1. A randomized testing framework is devised to evaluate the consistency between the thread-block-to-SM scheduling predictor and the actual scheduler in NVIDIA GPGPUs. Initially, a round-robin scheduling predictor is implemented. Subsequently, continuous comparisons are conducted between the scheduling predictions and the actual GPGPU scheduling results on randomly generated concurrent kernel sequences. In instances where discrepancies emerge between predictions and actual results, an analysis of NVIDIA GPGPU's implicit scheduling mechanisms pertinent to the specific concurrent kernel sequence is conducted, and corresponding updates are made to the scheduling predictor, which constitutes an iterative process aimed at progressively aligning the scheduling predictor with the NVIDIA GPGPU thread block scheduler.

Through iterative analysis and updates, we have successfully reconstructed the precise and detailed thread block-to-SM scheduling algorithm utilized in NVIDIA GPGPUs. The scheduling predictor exhibits complete alignment with the actual scheduling results of NVIDIA GPGPUs and passes rigorous randomized testing. During the iterative refinement process, a comprehensive exploration is conducted into the NVIDIA GPGPU thread block-to-SM scheduling mechanisms as well as the SM resource management mechanisms. Beyond the resource-aware scheduling mechanism, this work also elucidates several significant scheduling and resource management mechanisms intrinsic to NVIDIA GPGPUs. These mechanisms encompass the warp-to-SM processing block scheduling mechanism, shared memory reconfiguration mechanism, and register renaming mechanism, which are proposed for the first time to the best of our knowledge. This contribution facilitates accurate performance modeling of GPGPUs as well as provides programmers with insights for potential performance optimization by performing fine-grained manipulation on CUDA programs involving concurrent kernels.

The rest of this paper is organized as follows: Section II introduces the background knowledge of NVIDIA GPGPUs, the CUDA programming model, and thread block scheduling. Section III exhibits the methodology, particularly the randomized testing framework employed to dissect NVIDIA GPGPU thread block-to-SM scheduling mechanisms. Section IV analyzes several significant mechanisms related to thread block-to-SM scheduling and SM resource management. Section V demonstrates the comprehensive NVIDIA GPGPU thread block-to-SM scheduling algorithm as well as an overview of SM resource management. Section VI is the conclusion.

II. Background

A. NVIDIA GPGPU Hardware Architecture

NVIDIA GPGPUs are parallel processors based on the SIMT architecture, designed for general-purpose computing workloads. Recent Volta, Turing, and Ampere GPGPUs share a similar hardware architecture [20]–[23]. Fig. 2 illustrates the architecture of NVIDIA GPGPUs. A single GPGPU interacts with memory through the memory controllers and is connected to other processors via NVLink or peripheral component interconnect express (PCIe) buses. The GPGPU features a shared L2 cache and multiple SMs organized by graphic processing clusters (GPCs) and texture processing clusters (TPCs). Each GPC contains multiple TPCs, and each TPC contains two SMs. Each SM is divided into four SM processing blocks, which share L1 cache and shared memory. Each SM processing block has its own pipeline and executes instructions at the warp (i.e. 32 threads) level. The SM processing blocks have their own register file, CUDA cores, tensor core(s), and load/store units.

As for the memory hierarchy of NVIDIA GPGPUs [24], [25], the memory and L2 data/constant/instruction cache are shared across the entire GPGPU, while the L1 data cache/shared memory, L1.5 constant cache/L1 instruction cache, and L1 constant cache are private to each SM. The register file and L0 instruction cache are private to each SM

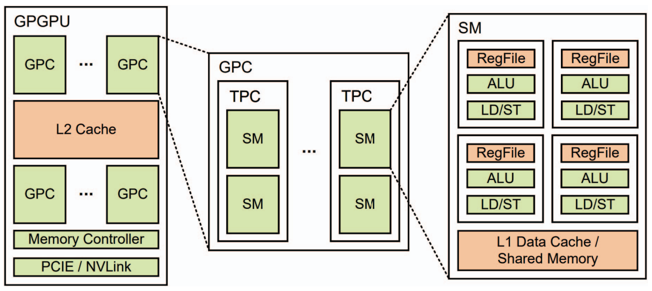

Fig. 2. Architecture of NVIDIA GPGPUs.

TABLE I
SM RESOURCE LIMITATIONS OF RTX3090.

GPGPU	RTX3090
Architecture	Ampere
Compute Capability	8.6
GPCs	7
TPCs	41
SMs	82
Max Thread Blocks / SM	16
Max Threads / SM	1536
Max Threads / Thread Block	1024
Max Registers / Thread	255
Register File / SM	65536 $\times$ 32 bit-registers
Register Allocation Granularity	32 threads $\times$ 8 registers
L1 Data Cache + Shared Memory / SM	128 KB
Shared Memory / SM Configuration	8/16/32/64/100 KB
Shared Memory Allocation Granularity	128 B
Runtime Shared Memory / Thread Block	1 KB

processing block. Notably, the unified L1 data cache and shared memory share the same static random access memory (SRAM) and can be configured into different combinations of L1 data cache and shared memory sizes.

B. *CUDA Programming Model*

The CUDA programming model [26] enables programmers to define functions executed on GPGPUs called kernels. Unlike regular C++ code, a kernel can be executed in parallel by multiple threads. Threads are organized into thread blocks, and thread blocks are then organized into a grid. A grid is executed across the entire GPGPU, while each thread block is executed in one SM. Thread blocks are partitioned into warps, where each warp is executed in one SM processing block. This programming model defines the parallelism of different threads in a kernel, while the mechanism of parallelism between different kernels is enabled by CUDA streams. Kernels in the same stream must be executed in sequence, while kernels in different streams can be executed concurrently.

C. *SM Resource Limitations*

The workloads that can be accommodated simultaneously in a single SM are limited by the resources available in the SM, including thread block slots, warp slots, register files, and shared memory. Table I presents the resource limitations of a RTX3090 GPGPU provided by the NVIDIA Nsight Compute Occupancy Calculator [27].

By analyzing the resource usage of each thread block and the SM resource limitations of the GPGPU, the maximum number of thread blocks in a kernel that can be accommodated simultaneously in a single SM, which is also known as occupancy, can be determined. Considering a kernel with 512 threads per thread block, 32 registers per thread, and 32 KB shared memory per thread block is launched on a RTX3090 GPGPU, it can be calculated from Table I that a maximum of three thread blocks can be accommodated in a SM. In this case, the bottlenecks are warp slots and shared memory. When thread blocks with different resource usages from concurrent kernels are executed in the same SM, the monitoring and allocation of SM resources become more complicated, which will be further discussed in Section IV and Section V.

D. *Related Work*

In recent years, considerable efforts have been directed towards dissecting the internal mechanisms of NVIDIA GPGPUs [24], [25], [28]–[33]. Employing specialized testing methodologies, these works have conducted detailed analyses of memory hierarchy, computational units, and new features of each architecture. These works have greatly enriched the understanding of NVIDIA GPGPU architectures as well as their performance modeling and optimization. However, an aspect that has not been sufficiently explored is thread block scheduling mechanisms.

The leftover policy [7], [34] dictates that GPGPU resources should be allocated to a kernel in a prioritized manner, such that resources are dedicated solely to that kernel until its scheduling completes. Previous work [18] explores the overall scheduling hierarchy in NVIDIA GPGPUs, particularly the thread block-to-SM scheduling constrained by maximum threads per SM. The most room policy [19] further proposes thread block placement policies that prioritize scheduling thread blocks onto the SM capable of accommodating the highest number. Nevertheless, these investigations lack a comprehensive analysis of thread block-to-SM scheduling mechanisms as well as a precise and validated scheduling algorithm capable of addressing all scenarios. While acknowledging the relationship between thread block scheduling and SM resource limitations, the dissections of SM resource monitoring and allocation in these works are also not comprehensive. Consequently, scheduling algorithms derived solely from these insights may not consistently align with the actual scheduling results of NVIDIA GPGPUs.

Advancing beyond preceding studies, this work takes a step further by presenting a comprehensive and precise NVIDIA GPGPU thread block-to-SM scheduling algorithm. It dissects key mechanisms underlying thread block scheduling and SM resource management. A randomized testing framework is devised to comprehensively validate the correctness of the algorithm, confirming its alignment with the thread block-to-SM scheduler in NVIDIA GPGPUs.

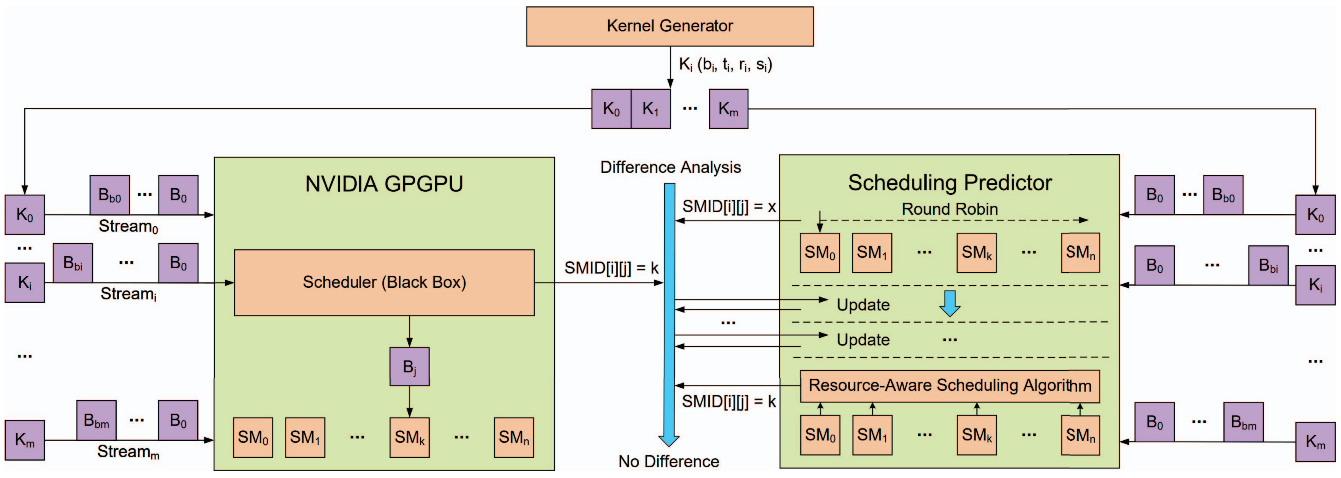

Fig. 3. Randomized Testing Framework.

III. Methodology

A randomized testing framework is developed to comprehensively analyze and validate the thread block-to-SM scheduling algorithm of NVIDIA GPGPUs. In this framework, diverse kernels with varying resource usages are generated and executed concurrently on a NVIDIA GPGPU, meanwhile the actual SM assignments for each thread block are recorded. To evaluate a specific scheduling algorithm, a thread block-to-SM scheduling predictor implementing the algorithm predicts the SM assignments for each thread block. By comparing the predicted SM identifications (SMIDs) with the actual SMIDs, the accuracy of the scheduling algorithm implemented by the predictor can be verified. When differences exist between the predictions and actual outcomes, the observed patterns in the actual thread block scheduling results guide updates to the scheduling predictor. This iterative process continues until the predictions always align perfectly with the actual GPGPU scheduling results, thus ultimately dissecting the thread block-to-SM scheduling mechanisms of NVIDIA GPGPUs. The framework is implemented on a NVIDIA RTX3090 server with CUDA 11.7.

A. Randomized Testing Framework

The randomized testing framework is illustrated in Fig. 3. The framework consists of three key components: a kernel generator, a NVIDIA GPGPU, and a thread block-to-SM scheduling predictor.

The kernel generator is designed to generate random kernel sequences. The parameters of each kernel, including the number of thread blocks, the number of threads, register usage, and shared memory usage, are all randomized within the bounds of GPGPU limitations. These random kernels are continuously generated and dispatched to the GPGPU for concurrent execution across distinct CUDA streams. This process persists until the GPGPU reaches its capacity and cannot accommodate the next kernel concurrently. The kernels are scheduled sequentially on the GPGPU, with the assignment of each thread block to a specific SM scheduled by the thread block scheduler. The SMID corresponding to the SM where each thread block is executed can be obtained.

Meanwhile, the scheduling predictor implemented based on a specific scheduling algorithm attempts to forecast the SM assignments for each thread block in the kernel sequence. The predictor is initiated in a round-robin manner. During the randomized testing, a manual difference analysis between the actual GPGPU scheduling SMIDs and the scheduling predictions is conducted. By analyzing these instances, significant mechanisms of NVIDIA GPGPU thread block scheduling and resource management are dissected, which will be further demonstrated in Section IV. Based on the dissected mechanisms, the scheduling predictor is iteratively updated. This iterative refinement process gradually adapts the thread block-to-SM scheduling algorithm of the predictor to mirror that of the NVIDIA GPGPU. Finally, this adaptation ensures a consistent alignment, revealing the complete NVIDIA GPGPU thread block-to-SM scheduling algorithm, which will be exhibited in Section V.

B. Kernel Code Design

The CUDA kernel utilized in the randomized testing framework necessitates meticulous design, especially focusing on two essential characteristics:

- Configurability. It should be flexible to configure the kernel scale and resource utilization, including the number of thread blocks, threads per thread block, register usage per thread, and shared memory usage per thread block.
- SMID Accessibility. The kernel needs to obtain the corresponding SMID for each thread block.

To address these requirements, the kernel code depicted in Listing 1 is devised. The number of thread blocks, threads, and shared memory usage can be configured during invocation. The main challenge lies in configuring register usage. Fortunately, the CUDA compiler nvcc supports the

Listing 1. Kernel Code.

```
__global__ void kernel(
    int *smid, int smem_size, float *seconds, clock_t clock_rate)
{
    clock_t t0 = clock64();

    int this_smid;
    asm("mov.u32 %0, %smid;" : "=r"(this_smid));
    if (threadIdx.x == 0) smid[blockIdx.x] = this_smid;

    extern __shared__ int smem[];
    const int regs_count = 256;
    const int smem_count = smem_size / sizeof(int);
    int regs[regs_count];

    #pragma unroll 256
    for (int i = 0; i < smem_count; i++)
        regs[i % regs_count] += smem[i];
    for (int i = 0; i < regs_count && i < smem_count; i++)
        smem[i] = regs[i];

    float this_seconds = seconds[blockIdx.x];
    clock_t t1 = clock64();
    while ((t1 - t0) / (clock_rate * 1000.0f) < this_seconds)
        t1 = clock64();
}
```

"`-maxrregcount`" option to limit register usage. Therefore, the kernel code is intentionally designed to utilize an excess of 256 registers and compel loop unroolling to avoid compiler optimizations that might diminish register usage. The "`-maxrregcount=n`" flag is applied during compilation to generate a kernel using almost precisely n registers.

Since the register allocation granularity is 8 registers per thread within a warp, 30 kernels are compiled with register usage ranging from 24, 32, 40, $\cdots$, 248, to 255. These kernels can be invoked by the kernel generator with a specified register count. It is essential to acknowledge that this kernel design entails certain trade-offs. Utilizing an excessive number of registers and constraining the register usage through compilation flags results in fluctuating local memory usage across the compiled kernels. In instances where register usage is reduced, local memory usage increases proportionally. Consequently, the kernel sequences generated by the kernel generator must ensure that the local memory usage of the first kernel equals or surpasses that of the subsequent kernels. This precaution prevents any hindrance to subsequent kernel executions due to GPGPU local memory reconfiguration.

Moreover, the parallel thread execution (PTX) assembly instruction "`mov.u32 %0, %smid;`" is harnessed to obtain the SMID of the SM where each thread block is executed. Additionally, the "`clock64()`" function is used to control the execution time of each thread block for a certain duration.

IV. Key mechanisms

In this section, attention is directed towards several significant mechanisms associated with the NVIDIA GPGPU thread block-to-SM scheduling. In order to dissect each mechanism, a large number of complicated kernel sequences are tested, and the actual scheduling results are manually analyzed, which is a process difficult to demonstrate in the paper. Therefore, the analysis conclusions are listed directly, accompanied by the corresponding and easy-to-understand cases that substantiate these findings. These cases and their actual scheduling results are concisely listed in Table II. Parameters in these cases are given according to the specifications of the RTX3090 GPGPU in Table I. It's worth noting that the conclusions drawn from these cases apply to other Ampere GPGPUs as well, requiring parameter adjustments during the testing process.

A. Resource-Aware Scheduling Mechanism

Mechanism 1. NVIDIA GPGPUs monitor and allocate the SM resources in the thread block-to-SM scheduler. When scheduling a thread block, the scheduler chooses the SM capable of accommodating the maximum number of the current thread block. This selection is based on the real-time availability of thread block slots, warp slots, register files, and shared memory within each SM. In scenarios where multiple SMs are equally suitable for accommodating the current thread block, the scheduler prioritizes selection following the SMID order of $0, 2, \cdots, \frac{N\_SM}{2}-1, 1, 3, \cdots, \frac{N\_SM}{2}$, where N_SM represents the total number of SMs in the GPGPU.

In Case 1.1, after the scheduling of Kernel 1 and Kernel 2, both SM0 and SM1 can each accommodate $\lfloor\frac{4\times12-\lfloor256/32\rfloor}{\lfloor256/32\rfloor}\rfloor = 5$ thread blocks of Kernel 3 due to warp slot constraints. Consequently, the single thread block from Kernel 3 is sequentially scheduled to SM0.

In Case 1.2, the limited warp slots allow SM0 to accommodate $\lfloor\frac{4\times12-\lfloor512/32\rfloor}{\lfloor256/32\rfloor}\rfloor = 4$ more thread blocks of Kernel 3, while SM1 can accommodate $\lfloor\frac{4\times12-\lfloor256/32\rfloor}{\lfloor256/32\rfloor}\rfloor = 5$ thread blocks. In Case 1.3, the bottleneck shifts to register file, resulting in that SM0 can accommodate only $\lfloor\frac{65536-256\times128}{256\times64}\rfloor = 2$ thread blocks of Kernel 3 and SM1 can accommodate $\lfloor\frac{65536-256\times64}{256\times64}\rfloor = 3$ thread blocks. In Case 1.4, the limited shared memory capacity permits SM0 to accommodate $\lfloor\frac{100-(48+1)}{24+1}\rfloor = 2$ thread blocks of Kernel 3, while SM1 can accommodate $\lfloor\frac{100-(24+1)}{24+1}\rfloor = 3$ thread blocks.

Therefore, in Cases 1.2, 1.3, and 1.4, the single thread block of Kernel 3 is scheduled to SM1 rather than SM0. This preference of SM1 highlights that the NVIDIA GPGPU thread block-to-SM scheduler transcends mere round-robin scheduling. It instead implements an SM resource-aware scheduling algorithm, giving equal priority to thread block slots, warp slots, registers, and shared memory.

B. Warp-To-SM Processing Block Scheduling Mechanism

Mechanism 2.1. When scheduling each warp of a thread block to SM processing blocks, the NVIDIA GPGPU thread block scheduler employs a "strict" round-robin mechanism.

In this context, the term "strict" indicates that even if the current round-robin scheduler points to an processing block lacking idle resources for a new warp and other processing blocks within the same SM have enough space, the scheduler will not redirect the warp to another processing block. Instead, it will wait for the current processing block to free up space.

TABLE II
CASES TO SUBSTANTIATE THE DISSECTED MECHANISMS.

Case	Kernel	Thread Blocks	Threads	Registers	Shared Memory	Duration	Scheduling Results
Case 1.1	Kernel 1	41	256	32	0	1s	SMID = $0, 2, 4, \cdots, 40$.
	Kernel 2	41	256	32	0	1s	SMID = $1, 3, 5, \cdots, 41$.
	Kernel 3	1	256	32	0	1s	**SMID = 0**.
	* Kernel 1/2/3 can be executed concurrently.						
Case 1.2	Kernel 1	41	**512**	32	0	1s	SMID = $0, 2, 4, \cdots, 40$.
	Kernel 2	41	**256**	32	0	1s	SMID = $1, 3, 5, \cdots, 41$.
	Kernel 3	1	**256**	32	0	1s	**SMID = 1**.
	* Kernel 1/2/3 can be executed concurrently.						
Case 1.3	Kernel 1	41	256	**128**	0	1s	SMID = $0, 2, 4, \cdots, 40$.
	Kernel 2	41	256	**64**	0	1s	SMID = $1, 3, 5, \cdots, 41$.
	Kernel 3	1	256	**64**	0	1s	**SMID = 1**.
	* Kernel 1/2/3 can be executed concurrently.						
Case 1.4	Kernel 1	41	256	32	**48 KB**	1s	SMID = $0, 2, 4, \cdots, 40$.
	Kernel 2	41	256	32	**24 KB**	1s	SMID = $1, 3, 5, \cdots, 41$.
	Kernel 3	1	256	32	**24 KB**	1s	**SMID = 1**.
	* Kernel 1/2/3 can be executed concurrently.						
Case 2.1	Kernel 1/3	82	**64**	255	0	**2s**	SMID = $0, 1, 2, \cdots, 81$.
	Kernel 2/4	82	**64**	255	0	**1s**	SMID = $0, 1, 2, \cdots, 81$.
	Kernel 5	82	**128**	255	0	1s	
	* Kernel 5 **cannot** be executed concurrently with Kernel 1/3 after Kernel 2/4 exit.						
Case 2.2	Kernel 1	82	**32**	255	0	1s	SMID = $0, 1, 2, \cdots, 81$.
	Kernel 2	82	**128**	255	0	1s	SMID = $0, 1, 2, \cdots, 81$.
	Kernel 3	82	**96**	255	0	1s	
	* Kernel 3 **cannot** be executed concurrently with Kernel 1/2.						
Case 3	Kernel 1	41	1	32	**0**	1s	SMID = $0, 2, 4, \cdots, 80$.
	Kernel 2	1	1	32	**1 KB**	1s	
	* Kernel 2 **cannot** be executed concurrently with Kernel 1.						
Case 4.1	Kernel 1/3/5/7	82	128	**64**	0	**2s**	SMID = $0, 1, 2, \cdots, 81$.
	Kernel 2/4/6/8	82	128	**64**	0	**1s**	SMID = $0, 1, 2, \cdots, 81$.
	Kernel 9	82	128	**255**	0	1s	
	* Kernel 9 **can** be executed concurrently with Kernel 1/3/5/7 after Kernel 2/4/6/8 exit.						
Case 4.1	Kernel 1/3/5/7	82	32	32	**10 KB**	**2s**	SMID = $0, 1, 2, \cdots, 81$.
	Kernel 2/4/6/8	82	32	32	**10 KB**	**1s**	SMID = $0, 1, 2, \cdots, 81$.
	Kernel 9	82	32	32	**40 KB**	1s	
	* Kernel 9 **cannot** be executed concurrently with Kernel 1/3/5/7 after Kernel 2/4/6/8 exit.						

For instance, in Case 2.1, each kernel comprises 82 thread blocks, exactly one thread block per SM. With each thread utilizing 255 registers, a single SM processing block can accommodate two warps. Kernel 1/2/3/4 contains two warps per thread block, allowing simultaneous accommodation within a single SM. Kernel 2 and Kernel 4 are set to exit at $t = 1$, while Kernel 1 and Kernel 3 are set to exit at $t = 2$. Profiling results from the NVIDIA Nsight System reveal that despite Kernel 1 and Kernel 3 exiting at $t = 1$ and freeing up the resources needed for the execution of Kernel 5, Kernel 5 is not immediately scheduled. Instead, scheduling occurs only after the completion of Kernel 2 and Kernel 4 at $t = 2$.

The phenomenon observed in Case 2.1 arises from the "strict" round-robin warp-to-SM processing block mechanism. As illustrated in Fig. 4, at $t = 0$, Kernel 1/2/3/4 start execution, with the round-robin pointer indicating the subsequent warp to be scheduled to processing block 0. At $t = 1$, although the completion of Kernel 2 and Kernel 4 releases processing blocks 2 and 3, the "strict" round-robin mechanism results in that the SM cannot immediately accommodate an additional thread block of Kernel 5. Only at $t = 2$, once Kernel 1 and Kernel 3 have also exited, are the thread blocks of Kernel 5 scheduled for execution.

Mechanism 2.2. When the number of warps in a scheduled thread block is divisible by 4 (processing blocks per SM), an additional processing block is skipped. This means that the pointer in the round-robin algorithm, which indicates the next scheduled processing block, moves an extra step.

Case 2.2 exhibits the evidence. As shown in Fig. 5, a warp from the thread block of Kernel 1 is dispatched to processing block 0 at first. Following this, four warps from the thread block of Kernel 2 are scheduled to processing blocks 1, 2, 3, and 0, respectively. Ordinarily, using the round-robin approach, the three warps from the thread block of Kernel 3 would conventionally be scheduled to processing blocks 1, 2, and 3. However, due to Mechanism 2.2, these warps from Kernel 3 attempt to find placement in processing blocks 2, 3, and 0. Consequently, after accommodating one thread block of Kernel 1 and Kernel 2 each, the SM struggles to find room for the thread block of Kernel 3. Profiling results indicate that Kernel 3 starts execution only once Kernel 1 and Kernel 2 have both completed their execution. On the other hand, if each thread block of Kernel 3 comprises only 2 warps, they can be comfortably assigned to processing blocks 2 and 3, enabling concurrent execution with Kernel 1 and Kernel 2.

Case 2.2 serves as an illustrative example of Mechanism

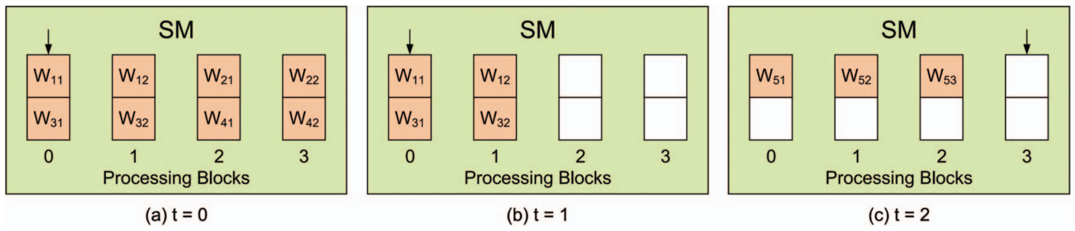

Fig. 4. Warp-To-SM Processing Block Scheduling of Case 2.1.

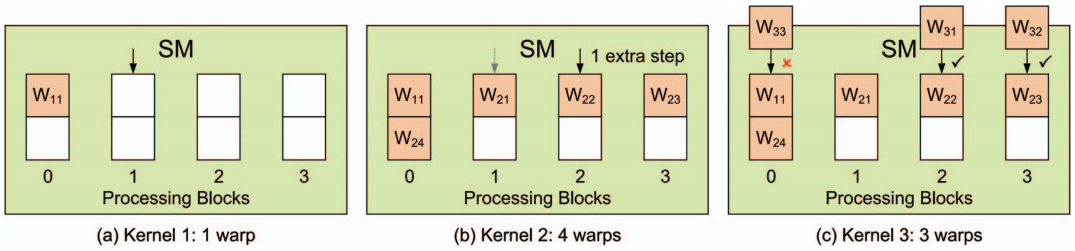

Fig. 5. Warp-To-SM Processing Block Scheduling of Case 2.2.

2.2. Rigorous testing within the randomized testing framework confirms that Mechanism 2.2 does indeed align with the actual scheduling behaviors of NVIDIA GPGPUs. However, the rationale behind Mechanism 2.2 remains elusive. This mechanism appears to underutilize the parallelism potential of GPGPUs in some situations. Actually, the strict round-robin mechanism of Mechanism 2.1 can also lead to suboptimal utilization of parallelism as illustrated in Case 2.1.

C. *Shared Memory Reconfiguration Mechanism*

Mechanism 3. If the shared memory configuration of a thread block exceeds the shared memory configuration of an SM, the thread block cannot be scheduled to that SM, even if there is sufficient shared memory available in that SM. The unified L1 data cache / shared memory configuration is established on a per-texture processing cluster (TPC) basis. The shared memory configuration of the two SMs within a TPC can only be set based on the shared memory configuration of the first thread block that enters when the TPC is completely empty. The shared memory configuration of a thread block is determined by multiplying the number of the thread block that an SM can accommodate by the total shared memory size used by the thread block. This calculated value is then rounded up to the nearest supported shared memory configuration, as listed in Table I for the RTX3090 GPGPU.

In Case 3, each SM is capable of accommodating 16 thread blocks of Kernel 1. Since each thread block of Kernel 1 occupies 1 KB shared memory (for CUDA runtime usage), the total shared memory demand for all 16 thread blocks amounts to 16 KB. Consequently, the shared memory configuration of Kernel 1 is 16 KB. Similarly, the shared memory configuration of Kernel 2 is 32 KB. This calculated shared memory configuration corresponds to the observed “Shared Memory executed” value in Nsight System. Given that all 41 thread blocks of Kernel 1 configure all 41 TPCs with a shared memory configuration of 16 KB, the thread blocks of Kernel 2 with a shared memory configuration of 32 KB must wait for the completion of Kernel 1 before they can be scheduled.

D. *Register Renaming Mechanism*

Mechanism 4. The allocation of physical registers to a warp by an SM processing block can be non-contiguous, whereas SM shared memory allocated to a thread block must be in contiguous space.

The logical registers used by the CUDA kernels need to be mapped to physical registers in the SM processing block through an injective function. To simplify the implementation, it is common practice to assign a physical register base address to each warp and map logical registers based on the offset from the base address. However, NVIDIA GPGPUs utilize a more complicated mapping method, which involves maintaining a mapping table between logical and physical registers for each warp at a specific allocation granularity, referred to as the register renaming mechanism. This method allows for the allocation of non-contiguous physical registers to compose larger logical register areas.

Case 4.1 provides evidence of the register renaming mechanism. After scheduling the first 8 kernels with each thread using 64 registers, all registers in the GPGPU are fully occupied. The odd-numbered kernels run for 1 second, while the even-numbered kernels run for 2 seconds. As a result, when odd-numbered kernels exit at $t = 1$, there are only four 64-register fragments available in each SM processing block to provide for the 256 registers required by each thread of kernel 9. Kernel 9 can be executed concurrently with Kernels 1/3/5/7, suggesting that each warp of Kernel 9 utilizes the four non-contiguous 64-physical-register fragments freed by Kernels 2/4/6/8, forming a 256-register logical register space.

In contrast to the register allocation mechanism, shared memory necessitates contiguous space. In Case 4.2, Kernel 9 cannot utilize the shared memory fragments released by Kernels 2/4/6/8 to fulfill its required shared memory space.

V. NVIDIA GPGPU Thread Block-To-SM Scheduler

A. Thread Block-To-SM scheduling Algorithm

The thread block-to-SM scheduling algorithm of NVIDIA GPGPUs dissected in this work is demonstrated in Algorithm 1. The algorithm takes the following inputs:

Algorithm 1: NVIDIA GPGPU Thread Block-To-SM Scheduling Algorithm.

```
Input: TB.t, TB.r, TB.s, TB.config_s, TB.config_l, GPGPU.gran_r,
       GPGPU.gran_s, GPGPU.cudart_s, GPGPU.config_l,
       SM[i].b, SM[i].TPC.config_s, SM[i].s, SM[i].p,
       SM[i].PB[j].w, SM[i].PB[j].r;
Output: A SMID, or CANNOT_SCHEDULE_NOW.
if not GPGPU.isIdle() and TB.config_l $>$ GPGPU.config_l then
    return CANNOT_SCHEDULE_NOW;
end
w $\leftarrow \lceil \frac{TB.t}{32} \rceil$;
r $\leftarrow \lceil \frac{TB.r \times 32}{GPGPU.gran\_r} \rceil \times$ GPGPU.gran_r;
s $\leftarrow \lceil \frac{TB.s}{GPGPU.gran\_s} \rceil \times$ GPGPU.gran_s $+$ GPGPU.cudart_s;
max_thread_blocks $\leftarrow$ 0;
simd $\leftarrow$ 0;
for i in $\{0, 2, 4, \cdots, N\_SM - 2, 1, 3, 5, \cdots, N\_SM - 1\}$ do
    if not SM[i].TPC.isIdle() and
      TB.config_s $>$ SM[i].TPC.config_s then
        continue;
    else
        limit_b $\leftarrow$ SM[i].b;
        limit_pb_w[4];
        for j in 0, 1, 2, 3 do
            limit_pb_w[j] $\leftarrow$ min(SM[i].PB[j].w, $\lfloor \frac{SM[i].PB[j].r}{r} \rfloor$);
        end
        for j in 0, 1, 2, 3 do
            smpb_p $\leftarrow$ (SM[i].p + j)%4;
            if limit_pb_w[smpb_p] == min(limit_pb_w) then
                limit_w $\leftarrow$ min(limit_pb_w) $\times 4 +$ j;
                break;
            end
        end
        limit_s $\leftarrow \lfloor \frac{SM[i].s}{s} \rfloor$ if not SM[i].TPC.isIdle() else
          $\lfloor \frac{TB.config\_s}{s} \rfloor$;
        cur_thread_blocks $\leftarrow$ min(limit_b, limit_w, limit_s);
        if cur_thread_blocks $>$ max_thread_blocks then
            max_thread_blocks $\leftarrow$ cur_thread_blocks;
            smid $\leftarrow$ i;
        end
    end
end
if max_thread_blocks == 0 then
    return CANNOT_SCHEDULE_NOW;
else
    return smid;
end
```

1) Information about the currently scheduled thread block: threads per thread block ($TB.t$), registers per thread ($TB.r$), shared memory per thread block ($TB.s$), shared memory configuration of the thread block ($TB.config\_s$), and local memory configuration of the thread block ($TB.config\_l$).
2) GPGPU hardware specifications: granularity of register allocation ($GPGPU.gran\_r$), granularity of shared memory allocation ($GPGPU.gran\_s$), shared memory per thread block used by CUDA runtime ($GPGPU.cudart\_s$).
3) The current resource status of the GPGPU: current local memory configuration ($GPGPU.config\_l$); resource status of the i-th SM, including available thread block slots ($SM[i].b$), current shared memory configuration of the TPC ($SM[i].TPC.config\_s$), availble contiguous shared memory ($SM[i].s$), and round-robin wrap scheduling pointer ($SM[i].p$); resource status of the j-th processing block of the i-th SM, including available warp slots ($SM[i].PB[j].w$) and available registers ($SM[i].PB[j].r$).

The main steps for determining the placement of a thread block in Algorithm 1 are listed as follows:

1) Verification of local memory configuration: determine whether the current local memory configuration of the GPGPU satisfies the requirements of the thread block. If not, the local memory configuration can only be adjusted when the GPGPU is completely idle.
2) Calculation SM capacity: calculate the required warp slots, registers, and shared memory for the current thread block. Based on the resource requirements and the available resources within each SM, calculate the maximum number of the current thread block that can be accommodated in each SM. While calculating, it is important to note:
   - Check if the shared memory configuration of the current SM meets the shared memory configuration of the thread block. If not, the shared memory configuration can only be reconfigured for both SMs when the TPC is completely idle.
   - After calculating the warps that each SM processing block can accommodate, the total warps the entire SM can accommodate may not equal the sum of warps that four processing blocks can accommodate. This calculation follows the warp-to-SM processing block scheduling mechanism described in Section IV-B.
3) SM selection: select the SM that can accommodate the maximum number of the current thread block for scheduling. In case of ties, prefer even-numbered SMs before odd-numbered ones.

The scheduling algorithm presented in Algorithm 1 is validated using the randomized testing framework demonstrated in Section III. The algorithm aligns perfectly with the actual scheduling results of NVIDIA GPGPUs.

B. SM Resource Management

The thread block scheduler of NVIDIA GPGPUs monitors the resources in each SM and allocates these resources when

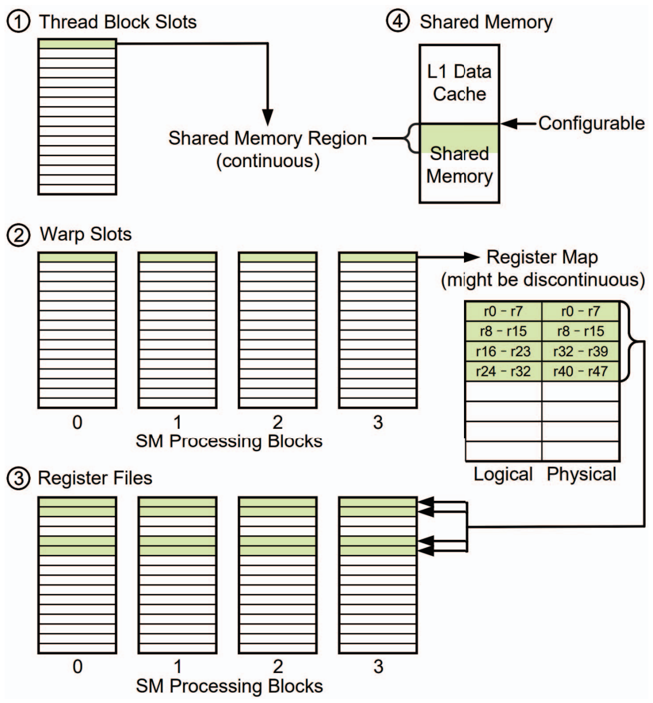

Fig. 6. SM Resource Management.

scheduling thread blocks. Fig. 6 illustrates the overall resource management approach for thread block slots, warp slots, register files, and shared memory within an SM.

Thread Block Slots and Warp Slots. The management of thread block slots and warp slots follows a straightforward approach. As discussed in section II-B, one or more warps are organized into a thread block. Even if a thread block contains only one warp, it still occupies one thread block slot in the SM. Similarly, even if a warp contains only one thread, it still occupies one warp slot in a specific SM processing block.

Register Files. The register file within a SM is distributed across four SM processing blocks and is privately owned by each of them. Each GPGPU with Volta, Turing, and Ampere architectures includes 65,536 registers in a SM, with 16,384 32-bit registers per SM processing block. Due to the access of registers within each SM processing block taking place at the granularity of warp (i.e., 32 threads), the register file is organized into multiple banks of SRAM, whose data width is 1024 bits. Each row of data in the SRAM corresponds exactly to one register for each of the 32 threads. The allocation granularity for registers is 8 registers per thread within a warp. Each CUDA kernel can utilize a maximum of 255 registers per thread, while each SM processing block can provide up to 512 registers per thread. The logical registers used by a CUDA kernel do not need to be contiguous within the physical register file of a processing block. Instead, a mapping table is maintained within each processing block to establish logical-to-physical register mapping for each warp.

Shared Memory. The size of shared memory is configurable. When a TPC is idle, the unified L1 data cache / shared memory in both SMs can be configured in a certain proportion. The shared memory space allocated to each thread block must be contiguous in physical space.

C. Discussion

Currently, NVIDIA GPGPU thread block-to-SM scheduling mechanisms effectively ensure a balanced distribution of workload across SMs, thereby optimizing resource utilization and facilitating concurrent execution of multiple thread blocks. However, certain factors continue to impact the achievement of higher levels of concurrency, particularly concerning warp-to-SM processing block scheduling and shared memory management.

To address these challenges, two primary recommendations emerge. Firstly, NVIDIA could refine their thread block scheduling strategies. For instance, they might consider relaxing the strict round-robin warp-to-SM processing block scheduling and introducing more flexibility to warp scheduling. Additionally, a more flexible approach to SM shared memory configuration can be adopted to enhance concurrency, where thread blocks can be admitted as long as adequate shared memory is available without the restriction on shared memory configuration.

Secondly, based on the existing NVIDIA GPGPU thread block scheduling mechanisms, when utilizing CUDA stream for concurrent kernel programming, meticulous attention should be given to the invocation order of kernels with distinct resource usage. Prioritizing kernels with a larger local memory configuration and a larger shared memory configuration can yield performance improvements. By assimilating insights from the NVIDIA GPGPU thread block-to-SM scheduling mechanisms dissected in this study, developers gain the ability to exercise more precise and fine-grained manual control over thread block scheduling.

VI. Conclusion

This work comprehensively dissects the NVIDIA GPGPU thread block-to-SM scheduling mechanisms and presents a precise NVIDIA GPGPU thread block-to-SM scheduling algorithm. Under random kernel sequences generated in the randomized testing framework, difference analyses of the scheduling predictions and the actual GPGPU scheduling results are conducted to dissect specific mechanisms and update the algorithm. After the iterative refinement of the scheduling algorithm, the scheduling predictions align perfectly with the actual scheduling results. Furthermore, this work thoroughly analyzes the details of SM resource management and elucidates the inherent connections between resource management and thread block scheduling. By dissecting the thread block scheduling and resource management mechanisms of NVIDIA GPGPUs, the placement of the scheduled thread blocks can be accurately predicted, which is particularly valuable for achieving precise modeling and potential performance optimizations of concurrent kernel execution on NVIDIA

GPGPUs. Future research will focus on the performance impact of thread block scheduling and investigate additional performance enhancements on actual workloads using the NVIDIA GPGPU thread block scheduling algorithm that has been studied and deconstructed in this work.

ACKNOWLEDGMENT

The authors thank the anonymous reviewers for their valuable comments and suggestions. This work is supported by the National Key Research and Development Program (Grant No.2022YFB3105103).

REFERENCES

[1] D. Kirk, "Nvidia cuda software and gpu parallel computing architecture," in *Proceedings of the 6th international symposium on Memory management (ISMM '07)*, 2007, pp. 103–104.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 26th Annual Conference on Neural Information Processing Systems (NIPS '12)*, 2012, pp. 1106–1114.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR '16)*, 2016, pp. 770–778.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st Annual Conference on Neural Information Processing Systems (NIPS '17)*, 2017, pp. 5998–6008.

[5] OpenAI, "GPT-4 technical report," Mar 2023, arXiv.2303.08774.

[6] C. Zhao, W. Gao, F. Nie, and H. Zhou, "A survey of gpu multitasking methods supported by hardware architecture," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 33, no. 6, pp. 1451–1463, Jun 2022.

[7] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram, "Warped-slicer: Efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*, 2016, pp. 230–242.

[8] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing," in *Proceedings of the 22nd International Symposium on High Performance Computer Architecture (HPCA '16)*, 2016, pp. 358–369.

[9] H. Zhao, W. Cui, Q. Chen, Y. Zhang, Y. Lu, C. Li, J. Leng, and M. Guo, "Tacker: Tensor-cuda core kernel fusion for improving the gpu utilization while ensuring qos," in *Proceedings of the 28nd International Symposium on High Performance Computer Architecture (HPCA '22)*, 2022, pp. 800–813.

[10] Z. Wang, J. Yang, R. G. Melhem, B. R. Childers, Y. Zhang, and M. Guo, "Quality of service support for fine-grained sharing on gpus," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*, 2017, pp. 269–281.

[11] S.-K. Shekofteh, H. Noori, M. Naghibzadeh, H. Fröning, and H. S. Yazdi, "ccuda: Effective co-scheduling of concurrent kernels on gpus," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 31, no. 4, pp. 766–778, Apr 2020.

[12] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving gpgpu resource utilization through alternative thread block scheduling," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA '14)*, 2014, pp. 260–271.

[13] A. Li, S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal, "Locality-aware cta clustering for modern gpus," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*, 2017, pp. 297—311.

[14] D. Tripathy, A. Abdolrashidi, L. N. Bhuyan, L. Zhou, and D. Wong, "Paver: Locality graph-based thread block scheduling for gpus," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 3, pp. 32:1–32:26, Jun 2021.

[15] M. Huzaifa, J. Alsop, A. Mahmoud, G. Salvador, M. D. Sinclair, and S. V. Adve, "Inter-kernel reuse-aware thread block scheduling," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 3, pp. 24:1–24:27, Aug 2020.

[16] A. Zou, J. Li, C. D. Gill, and X. Zhang, "Rtgpu: Real-time gpu scheduling of hard deadline parallel tasks with fine-grain utilization," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 34, no. 5, pp. 1450–1465, May 2023.

[17] H. Kim and W. J. Song, "Las: Locality-aware scheduling for gemm-accelerated convolutions in gpus," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 34, no. 5, pp. 1479–1494, May 2023.

[18] I. S. Olmedo, N. Capodieci, J. L. Martinez, A. Marongiu, and M. Bertogna, "Dissecting the cuda scheduling hierarchy: a performance and predictability perspective," in *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '20)*, 2020, pp. 213–225.

[19] G. Gilman, S. S. Ogden, T. Guo, and R. J. Walls, "Demystifying the placement policies of the nvidia gpu thread block scheduler for concurrent kernels," *ACM SIGMETRICS Performance Evaluation Review*, vol. 48, no. 3, pp. 81–88, Dec 2020.

[20] NVIDIA. (2017) Nvidia tesla v100 gpu architecture. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[21] NVIDIA. (2019) Nvidia turing gpu architecture. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[22] NVIDIA. (2020) Nvidia a100 tensor core gpu architecture. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[23] NVIDIA. (2020) Nvidia ampere ga102 gpu architecture. [Online]. Available: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

[24] X. Mei and X. Chu, "Dissecting gpu memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 1, pp. 72–86, Jan 2017.

[25] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," Apr 2018, arXiv.1804.06826.

[26] NVIDIA. (2023) Cuda c++ programming guide v12.1. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[27] NVIDIA. (2018) Nvidia nsight compute occupancy calculator. [Online]. Available: https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator

[28] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the nvidia turing t4 gpu via microbenchmarking," Mar 2019, arXiv.1903.07486.

[29] W. Sun, A. Li, T. Geng, S. Stuijk, and H. Corporaal, "Dissecting tensor cores via microbenchmarks: Latency, throughput and numeric behaviors," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 34, no. 1, pp. 246–261, Jan 2023.

[30] M. A. Raihan, N. Goli, and T. M. Aamodt, "Modeling deep learning accelerator enabled gpus," in *Proceedings of the 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '19)*, 2019, pp. 79–92.

[31] D. Yan, W. Wang, and X. Chu, "Demystifying tensor cores to optimize half-precision matrix multiply," in *Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS '20)*, 2020, pp. 634–643.

[32] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '10)*, 2010, pp. 235–246.

[33] H. Abdelkhalik, Y. Arafa, N. Santhi, and A. A. Badawy, "Demystifying the nvidia ampere architecture through microbenchmarking and instruction-level analysis," in *Proceedings of the 2022 IEEE High Performance Extreme Computing Conference (HPEC '22)*, 2022, pp. 1–8.

[34] B. W. Coon, J. R. Nickolls, J. E. Lindholm, R. J. Stoll, N. Wang, and J. H. Choquette, "Thread group scheduler for computing on a parallel thread processor," May 2014, US Patent NO.US8732713B2, Filed Sep 28th., 2011, Issued May 20th., 2014.